Cosa rende un design cattivo?

Sono quattro gli aspetti che rendono un design cattivo:

- Rigidità
- Fragilità
- Immobilità
- Viscosità

Per rigidità si intende un software difficile da modificare. Ogni modifica provoca una cascata di modifiche successive nei moduli dipendenti.

Un software è fragile quando tende a rompersi in molti punti ogni volta che viene modificato. Una modifica provoca comportamenti inaspettati in altre parti del sistema anche non concettualmente collegate alla parte modificata.

L'immobilità del software riguarda l'impossibilità di riutilizzare software di altri progetti o di parti del progetto stesso. In tal caso se uno sviluppatore ha bisogno di un modulo simile a quello scritto da un altro sviluppatore, ma questo modulo ha troppe dipendenze con altri moduli, il lavoro e il rischio necessari per separare le parti desiderate possono risultare eccessivi. Viene quindi riscritto un modulo molto simile dall'inizio.

Ci sono due tipi di viscosità:

- La viscosità del "design", si riferisce alla facilità con cui è possibile aggiungere/modificare codice software per supportare un cambiamento al progetto senza "intaccarne" il design. La viscosità del design è alta se è difficile effettuare un cambiamento con modifiche che mantengono "inalterato" il design, mentre è più facile effettuarlo attraverso modifiche che invece "alterano" il design esistente.
- La viscosità dell' "ambiente", si riferisce al processo di sviluppo e in particolare alla "dinamicità" e all'efficienza dell'ambiente di sviluppo. La viscosità dell'ambiente è alta se l'ambiente è lento e inefficiente. Ad esempio, se il processo di build è particolarmente lento, gli sviluppatori tenderanno ad effettuare le modifiche che prevedano il numero minimo di re-build, anche se queste modifiche non fossero le migliori dal punto di vista del design.

The Single Responsability Principle (SRP)

Nella programmazione orientata agli oggetti, il principio di singola responsabilità (single responsibility principle, abbreviato con SRP) afferma che ogni elemento di un programma (classe, metodo, variabile) deve avere una sola responsabilità, e che tale responsabilità debba essere interamente incapsulata dall'elemento stesso. Tutti i servizi offerti dall'elemento dovrebbero essere strettamente allineati a tale responsabilità.

Il principio prevede di sviluppare classi o moduli indipendenti tra loro e semplici.

Un esempio può essere il seguente:

- Si ha una classe **`Stampante`**, che possiede due funzionalità indipendenti tra loro, **`Stampa`** e **`Scan`**.
- Unire le due funzionalità in un'unica classe risulta poco portabile e la modifica ad una funzionalità influenza l'altra, quindi si creano due classi separate: **`StampanteSemplice`** e **`Scanner`**.

The Dependecy Inversion Principle (DIP)

Questo principio ci dice che ogni dipendenza dovrebbe puntare a un'interfaccia o a una classe astratta e nessuna dipendenza dovrebbe puntare a una classe concreta. I moduli di alto livello (i clienti) non dovrebbero dipendere dai moduli di basso livello (i fornitori di servizi), entrambi dovrebbero dipendere da astrazioni.

Le astrazioni inoltre non dovrebbero dipendere dai dettagli, viceversa i dettagli dovrebbero dipendere dalle astrazioni.

I moduli di basso livello contengono la maggior parte del codice e della logica implementativa e quindi sono i più soggetti a cambiamenti. Se i moduli di alto livello dipendono dai dettagli dei moduli di basso livello (sono accoppiati in modo troppo stretto), i cambiamenti si propagano e le conseguenze sono rigidità, fragilità e immobilità del software.

Questo principio funziona perché:

- Le astrazioni non contengono praticamente codice e quindi sono quasi esenti da cambiamenti.
- I moduli non astratti sono soggetti a cambiamenti, ma questi sono sicuri dal momento che da tali moduli non dipende nessuno.

Valgono dunque:

- Design for change: i dettagli del sistema sono isolati e i cambiamenti non si propagano.
- Design for reuse: i moduli sono maggiormente riutilizzabili poiché fortemente disaccoppiati.

Le astrazioni non dovranno mai essere modificate ma dovranno essere estese.

Un esempio può essere:

- Si considerino le classi concrete **`Addizione`**, **`Sottrazione`** e **`Calcolatrice`**; quest'ultima ha una dipendenza verso **`Addizione`** e **`Sottrazione`**.
- Tale dipendenza è verso una classi concrete, pertanto il codice di **`Calcolatrice`** è poco manutenibile. Infatti, nel caso si volesse creare la classe **`Moltiplicazione`** occorrerebbe modificare il codice di **`Calcolatrice`**.
- Per risolvere il problema, si introduce l'interfaccia **`IOperazione`** e si fa sì che **`Calcolatrice`** dipenda da essa; inoltre, si modificano le classi **`Addizione`** e **`Sottrazione`** in modo che implementino **`IOperazione`**.

- A questo punto, se si volesse aggiungere una nuova operazione, non sarebbe più necessario modificare **`Calcolatrice`** ma basterebbe creare una classe concreta che implementi **`IOperazione`** (ad esempio, **`Moltiplicazione`**)

The Interface Segregation Principle (ISP)

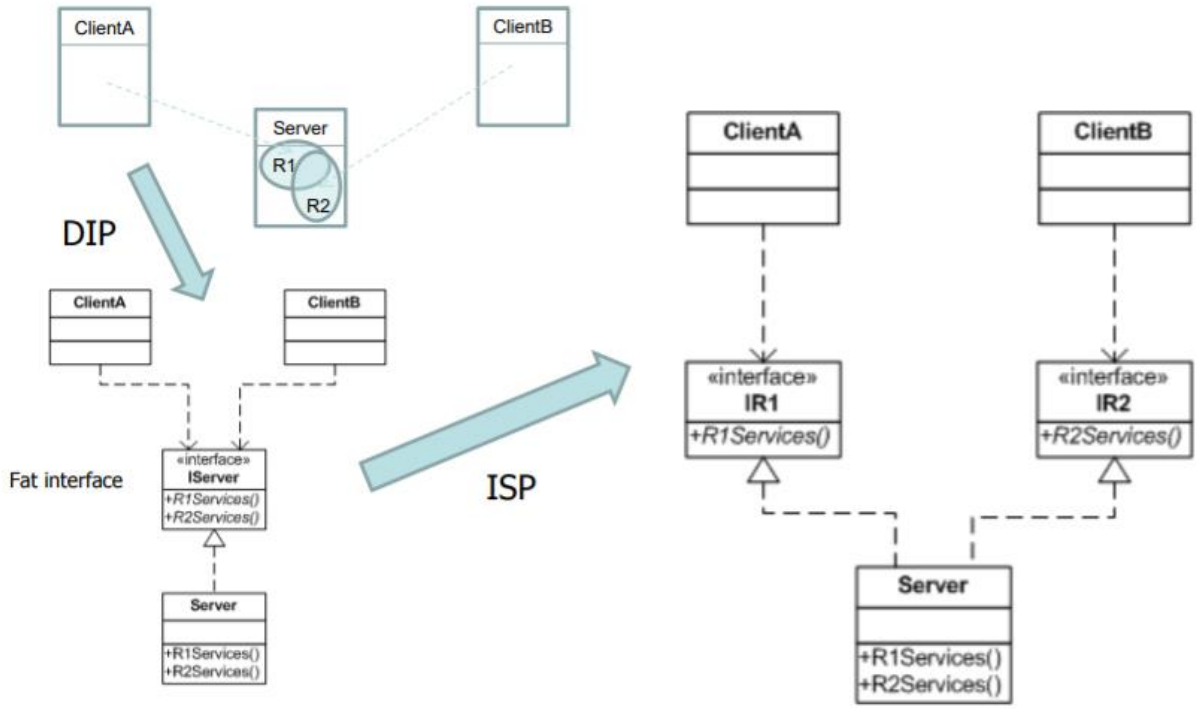
Questo principio afferma che molte interfacce per un cliente sono meglio di un'unica interfaccia general purpose.

I clienti non dovrebbero dipendere da servizi che non utilizzeranno (forma di accoppiamento indiretto inutile); se un cliente richiede l'aggiunta di una nuova funzionalità all'interfaccia allora ogni altro cliente sarà costretto a cambiare anche se non è interessato alla nuova funzionalità. Questo provoca uno sforzo eccessivo in fase di manutenzione e può portare a molti errori. Quindi non bisogna creare le "fat interface".

Si cerca quindi si cerca di suddividere i servizi in gruppi in modo da creare interfacce specifiche per ogni tipo di cliente e implementare tutte queste interfacce in una classe.

Un esempio potrebbe essere il seguente:

- Si ha l'interfaccia **`IStampante`** che presenta i metodi **`stampa()`** e **`scanner()`**.
- La classe concreta **`Stampante`** che implementa **`IStampante`**.
- Per l'Interface Segregation Principle è meglio separare l'interfaccia **`IStampante`** in due interfacce, **`IStampanteSemplice`** che definisce il metodo **`stampa()`** e **`IScanner`** che definisce il metodo **`scanner()`**.
- **`IStampanteSemplice`** e **`IScanner`** vengono implementate da **`Stampante`**.



The Open/Closed Principle (OCP)

Questo principio è fondamentale per la progettazione di entità riutilizzabili e afferma che le entità software come classi, moduli, funzioni, …, dovrebbero essere aperte a estensioni ma chiuse a modifiche.

Le entità open possono essere estese aggiungendo un nuovo stato o proprietà comportamentali. Mentre le entità close hanno un'interfaccia ben definita, pubblica e stabile che non può mai essere modificata. Nel caso in cui si vogliano aggiungere nuove funzionalità sarà necessario, anziché modificare il codice, creare una nuova classe concreta che implementi le astrazioni.

Per soddisfare il principio aperto/chiuso si occorre all'ereditarietà:

- Di interfaccia: le classi derivate ereditano da una classe base astratta con funzioni virtuali. In questo modo l'interfaccia è chiusa alle modifiche e il suo comportamento può essere modificato implementando nuove classi.
- Dell'implementazione: si creano nuove sottoclassi che estendono la classe base, mantenendo il codice comune in quest'ultima. In questo modo si evitano ripetizioni nelle sottoclassi.

Questo approccio consente una maggiore modularità del sistema e stabilità, in quanto non si modificano componenti definite precedentemente e non si introducono eventuali errori dovuti alle modifiche.

Un esempio di utilizzo è il seguente:

- È definita un'interfaccia `IPezzo` che rappresenta un pezzo degli scacchi. Quest'interfaccia definisce il metodo `muovi()`.
- Le classi concrete `Pedone` e `Alfiere` implementano `IPezzo` e possono ridefinire il metodo `muovi()`.
- In questo modo per aggiungere pezzi non è necessario modificare alcun codice, ma basta creare una classe concreta tipo `Cavallo` ed implementare `IPezzo`.

Se la maggior parte dei moduli di un'applicazione segue OCP, allora è possibile aggiungere nuove funzionalità all'applicazione aggiungendo nuovo codice, invece che cambiando codice funzionante. Il codice che già funziona non è esposto a rotture.

The Liskov Substitution Principle (LSP)

Questo principio afferma che se S è una sottoclasse di T, allora oggetti dichiarati di tipo T possono essere sostituiti con oggetti di tipo S senza alterare la correttezza del programma.

Questo significa che il client di una classe base deve continuare a funzionare correttamente se gli viene passato un sottotipo di tale classe base. In altre parole, un client che usa istanze della classe T devono poter utilizzare istanze della classe S (o di una qualunque sottoclasse di T) senza accorgersi delle differenze.

La principale violazione del principio di Liskov è data dalla ridefinizione di metodi virtuali nelle classi derivate. La chiave per evitare queste violazione è il design by contract dove ogni metodo ha:

- Un insieme di precondizioni (vincoli del chiamante): requisiti che devono essere soddisfatti dal chiamante perché il metodo possa essere eseguito correttamente.
- Un insieme di postcondizioni (vincoli del metodo): requisiti che devono essere soddisfatti dal metodo in caso di corretta esecuzione.

Precodizioni e postcondizioni costituiscono un contratto tra il client e il metodo.

Se il metodo viene ridefinito in una sottoclasse:

- Le precondizioni devono essere uguali o meno stringenti.
- Le postcondizioni devono essere uguali o più stringenti.

Un esempio tipico in cui viene violato il principio di Liskov è quello di una classe **`Quadrato`** che derivi da una classe **`Rettangolo`**. Si assume che i metodi setter e getter esistano per queste classi sia per la larghezza che per l'altezza. Per la classe **`Quadrato`** ci aspettiamo che larghezza ed altezza siano uguali. Se un oggetto **`Quadrato`** viene utilizzato in un contesto ove ci si aspetta sia usato un oggetto **`Rettangolo`**, si può verificare un comportamento inatteso perché le dimensioni (larghezza ed altezza) dell'oggetto **`Quadrato`** non possono e non dovrebbero essere modificate l'una indipendentemente dall'altra. Questo problema non può essere sistemato facilmente: se si modificano i metodi setter della classe **`Quadrato`** in modo tale che essi preservino le dimensioni uguali, allora questi metodi violeranno le postcondizioni per i metodi setter della classe Rettangolo, postcondizioni che affermano che per il rettangolo la larghezza e l'altezza possono variare in modo indipendente. Se **`Quadrato`** e **`Rettangolo`** hanno i soli metodi getter (cioè sono oggetti che non cambiano), non si verifica alcuna violazione del principio LSP.

Un altro esempio è il seguente:

- Data la classe **`Uccello`** che presenta il metodo **`vola()`**.
- La sottoclasse **`Gabbiano`** che estende **`Uccello`** non presenta una violazione del principio, in quanto la semantica del metodo **`vola()`** viene rispettata.
- La sottoclasse **`Gallina`** che estende **`Uccello`** viola il principio, in quanto questa classe non può ridefinire **`vola()`** e quindi non può essere utilizzata in sostituzione della classe base senza alterare la correttezza del programma.

Reuse/Release Equivalency Principle (REP)

Questo principio afferma che un elemento riutilizzabile, sia esso un componente, una classe o un insieme di classi, non può essere riutilizzato a meno che non sia gestito da un sistema di rilascio di qualche tipo.

I clienti dovrebbero rifiutare di riutilizzare un elemento a meno che l'autore non prometta di tenere traccia dei numeri di versione e di mantenere le vecchie versioni per qualche tempo.

Common Closure Principle (CCP)

Il lavoro per gestire, testare e rilasciare un pacchetto in un sistema di grandi dimensioni non è banale. Più pacchetti cambiano in un dato rilascio, maggiore è il lavoro per ricostruire, testare e distribuire il rilascio.

Vorremmo quindi ridurre al minimo il numero di pacchetti che vengono modificati in un dato ciclo di rilascio del prodotto. Per raggiungere questo obiettivo, raggruppiamo assieme classi che pensiamo cambieranno insieme

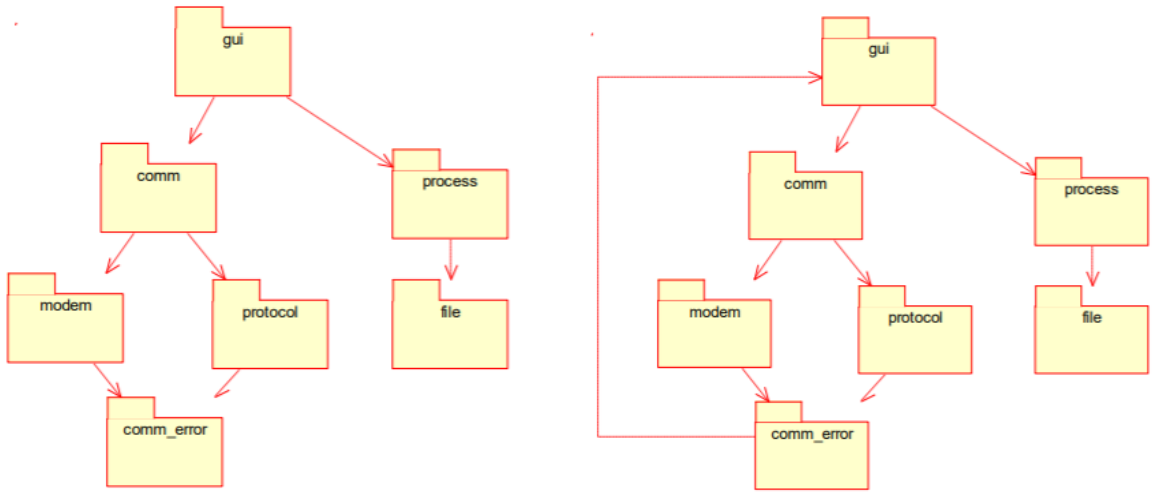
Common Reuse Principle (CRP)

Una dipendenza da un package è una dipendenza da tutto ciò che è contenuto nel package. Quando un package cambia e il suo numero di rilascio viene aggiornato, tutti i client di quel package devono verificare di funzionare con il nuovo package, anche se nulla di ciò che hanno usato all'interno del package è effettivamente cambiato. Pertanto, le classi che non vengono riutilizzate insieme non dovrebbero essere raggruppate insieme.

Acyclic Dependecies Principle (ADP)

Una volta apportate le modifiche a un package, gli sviluppatori possono rilasciare i package al resto del progetto. Prima di poter eseguire questo rilascio, tuttavia, devono verificare che il package funzioni e per farlo, devono compilarlo e collegarlo a tutti i package da cui dipende.

Una singola dipendenza ciclica che sfugge al controllo può rendere l'elenco delle dipendenze molto lungo, quindi, qualcuno deve osservare la struttura delle dipendenze dei package con regolarità e interrompere i cicli ovunque compaiano.

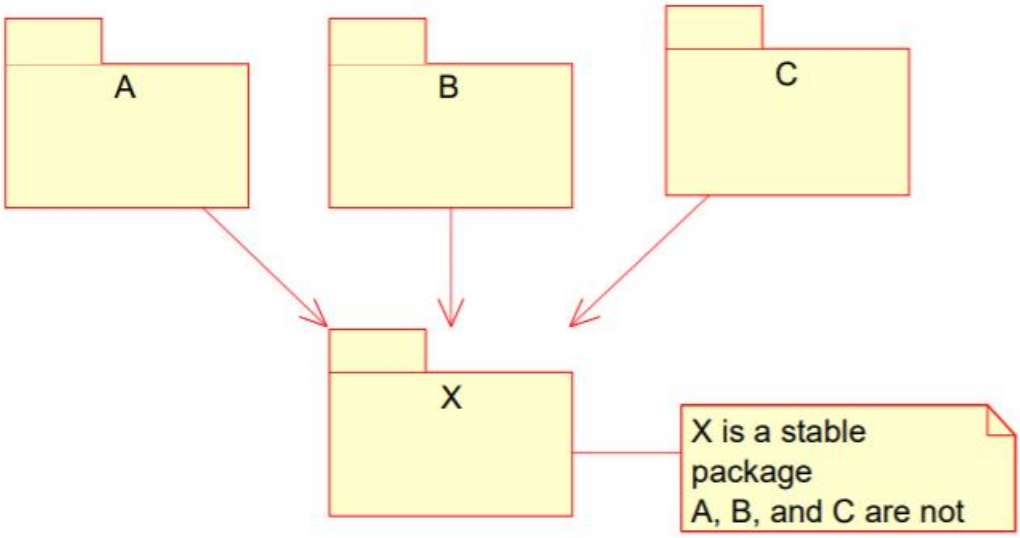


Nello scenario aciclico per rilasciare il package protocol, gli ingegneri dovrebbero compilarlo con l'ultima versione del package comm_error ed eseguire i loro test.

Nello scenario ciclico per rilasciare il package protocol, gli ingegneri dovrebbero compilarlo con l'ultima versione di comm_error, gui, comm, process, modem, file ed eseguire i loro test.

Stable Dependecies Principle (SDP)

I design non possono essere completamente statici. Una certa volatilità è necessaria se il progetto deve essere mantenuto. Alcuni package sono progettati per essere volatili, ci aspettiamo che cambino, un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare qualsiasi modifica con tutti i pacchetti dipendenti.



Stable Abstractions Principle (SAP)

La stabilità è relativa alla quantità di lavoro richiesta per apportare una modifica, un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare le modifiche con tutti i pacchetti dipendenti.

Cosa sono e come sono classificati i design pattern?

I design pattern servono a risolvere problemi progettuali specifici e in generale rendere i progetti object-oriented più flessibili e riutilizzabili.

Ogni design pattern ha quattro elementi essenziali:

- Nome: identifica il pattern.
- Problema: descrive quando applicare il pattern.
- Soluzione: descrive il pattern, cioè gli elementi che lo compongono, le loro relazioni e responsabilità.
- Conseguenze: vantaggi e svantaggi dell'applicazione del pattern.

Abbiamo 3 tipi di design pattern:

- Pattern di creazione: risolvono problemi inerenti il processo di creazione di oggetti.
- Pattern strutturali: risolvono problemi inerenti la composizione di classi o di oggetti.
- Pattern comportamentali: risolvono problemi inerenti la modalità di interazione e di distribuzione delle responsabilità tra classi o oggetti.

Pattern Singleton (di creazione)

Questo pattern assicura che una classe abbia una ed una sola istanza e fornisce un punto di accesso globale a tale istanza.

Questa classe deve:

- Tenere traccia della sua sola istanza.
- Intercettare tutte le richieste di creazione, al fine di garantire che nessuna altra istanza venga creata.
- Fornire un modo per accedere all'istanza.

```
public class Singleton {
     ...
     private static Singleton _instance = null;
     protected Singleton() { ... }
     public static Singleton GetInstance() {
          if(_instance == null)
               _instance = new Singleton();
          return _instance;
     }
     ...
}
```

Il costruttore è privato, in quanto assicura che viene creata al più un'istanza della classe `Singleton`. Il metodo statico `getInstance()` controlla se è già esistente un'istanza della classe: in caso affermativo restituisce l'istanza creata in precedenza, altrimenti crea una nuova istanza. Una classe con soli membri statici non rappresenta un'alternativa al pattern Singleton, in quanto non permette di creare istanze personalizzate in base al contesto; non permette inoltre di utilizzare un numero arbitrario di interfacce.

Un esempio di utilizzo del pattern Singleton è l'accesso a un database, in quanto si vuole garantire atomicità.

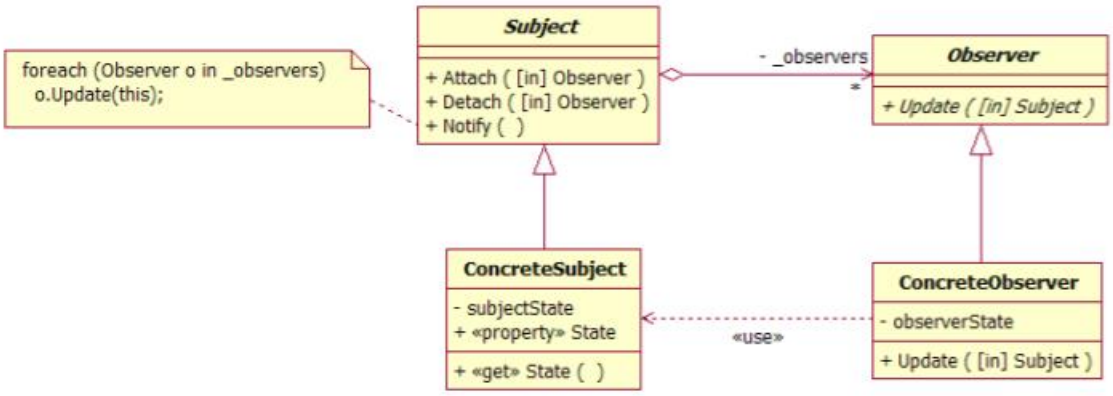
Pattern Observer (comportamentale)

Sostanzialmente il pattern si basa su uno o più oggetti, chiamati observer, che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto "osservato", che può essere chiamato soggetto.

Vengono quindi definite due classi:

- **`Subject`** (oggetto osservato): che dichiara i metodi **`Attach()`**, **`Detach()`** e **`Notify()`**.
- **`Observer`** (oggetti osservatore): che dichiara il metodo **`Update()`**.

Per registrarsi presso un **`Subject`** un **`Observer`** invoca il metodo **`Attach()`**. Successivamente in caso di evento **`Subject`** chiama il metodo **`Notify()`**, che esegue **`Update()`** di tutti gli **`Observer`** che si sono registrati. Infine un **`Observer`** si può disiscrivere da un **`Subject`** attraverso il metodo **`Detach()`**.



Quattro approcci alternativi sono:

- Class based: ogni subject ha un riferimento diretto all'observer da notificare.
- Interface based: ogni subject contiene un riferimento all'interfaccia implementata dagli observer, aumentando il disaccoppiamento tra subject e observer.
- Delegate based: ogni subject contiene un delegato pubblico in modo che gli observer possano registrarsi a tale delegato e venire notificati indirettamente dal subject.
- Event based: un event si occupa di effetture **`Detach()`** dagli observer.

Un esempio può essere il seguente:

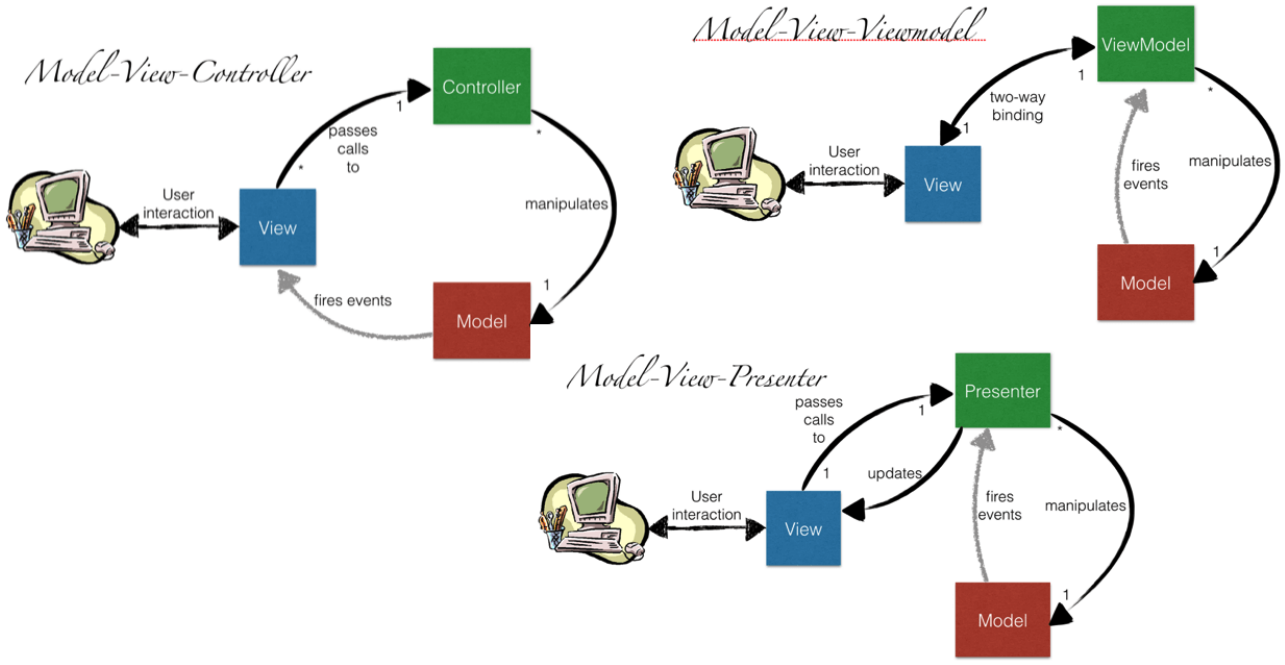
- Una classe **`Sensore`** (ad esempio temperatura, umidità, …) che rappresenta il **`Subject`**.
- La classe **`Stazione`** che rappresenta l'**`Observer`** verrà avvertita quando, ad esempio, il **`Sensore`** rileva una determinata temperatura.

Pattern MVC e MVP (architetturale)

Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- Il model fornisce i metodi per accedere ai dati utili all'applicazione;
- La view visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
- Il controller riceve i comandi dell'utente (in genere attraverso la view) e li attua modificando lo stato degli altri due componenti.

Questo schema, fra l'altro, implica anche la tradizionale separazione fra la logica di business, a carico del controller e del model, e l'interfaccia utente a carico del view.



Pattern Flyweight (strutturale)

Questo pattern descrive come condividere oggetti leggeri, in modo tale che il loro uso non sia troppo costoso.

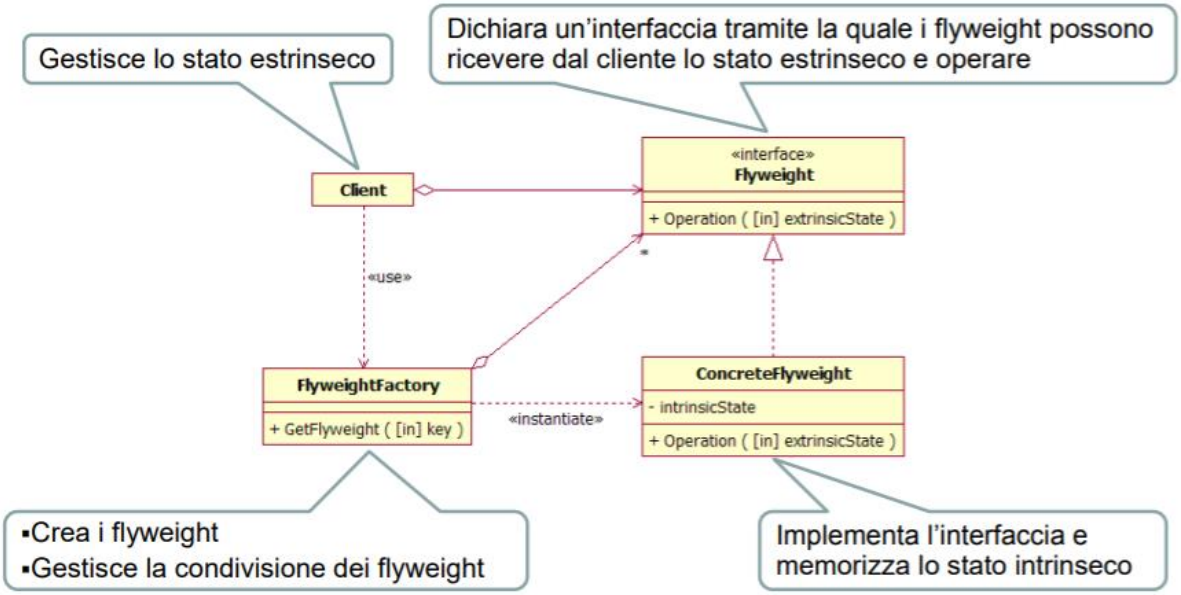
Un flyweight è un oggetto condiviso che può essere utilizzato simultaneamente ed efficientemente da più clienti fra loro indipendenti.

Nonostante sia condiviso deve essere indistinguibile da un oggetto non condiviso. I clienti:

- Non devono poter istanziare direttamente un flyweight.
- Devono poter ottenere l'oggetto da una FlyweightFactory.

Esistono due stati associati al pattern:

- Intrinseco: stato memorizzato all'interno del flyweight e non dipende dal contesto di utilizzo.
- Estrinseco: stato memorizzato dal cliente e passato ogni volta al flyweight quando è necessario invocare un'operazione relativa al flyweight stesso.



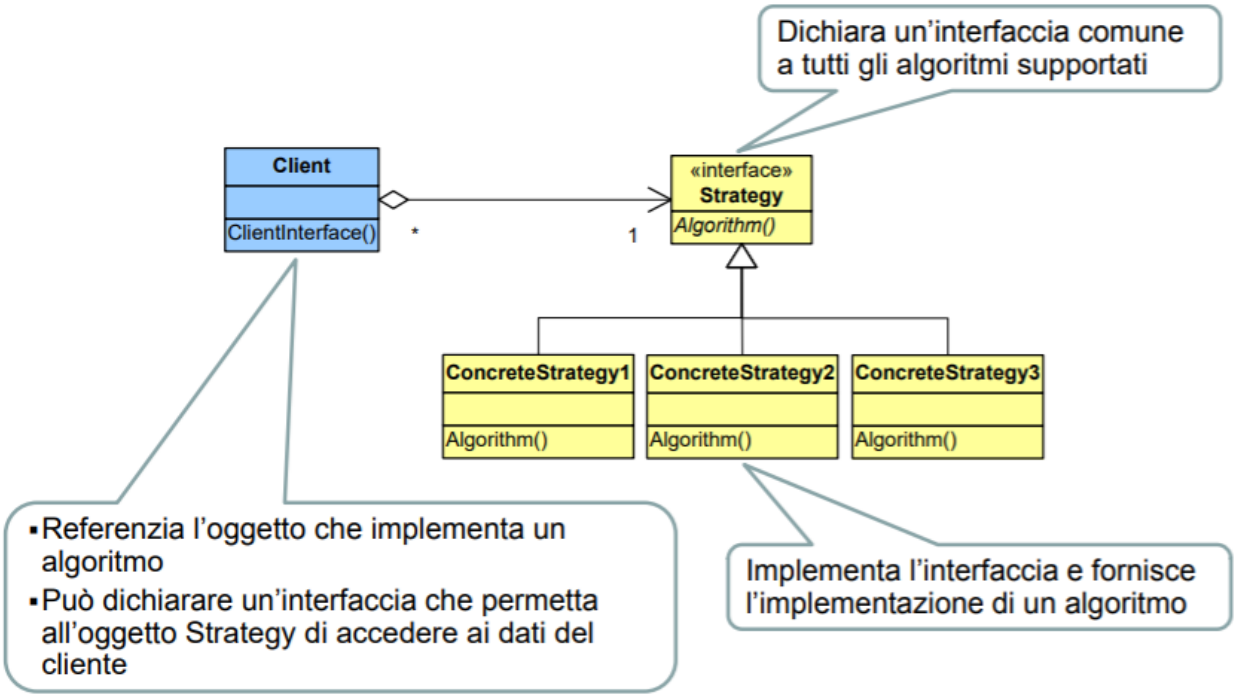
Un esempio può essere una cella di una griglia:

- Si consideri la classe `Cella`, senza pattern flyweight ogni cella della griglia risulterebbe un nuovo oggetto `Cella`, causando un'alta occupazione di memoria.
- Applicando patter flyweight si riutilizza la stessa cella senza ridefinire un nuovo oggetto ad ogni sua occorrenza.
- La classe `Cella` come stato intrinseco può avere ad esempio una dimensione e un colore. Mentre come stato estrinseco mantenuto dal cliente può avere la posizione (riga, colonna).

Pattern Strategy (comportamentale)

Questo pattern permette di:

- Definire un insieme di algoritmi tra loro correlati.
- Incapsulare tali algoritmi in una gerarchia di classi.
- Rendere gli algoritmi intercambiabili.



Questo pattern può essere utilizzato immaginando di avere un pezzo degli scacchi, ad esempio la regina, e rappresentare il suo movimento:

- Abbiamo il client **`Regina`**, l'interfaccia **`Strategy`** che espone il metodo **`movimento()`**.
- L'interfaccia può essere implementata da diverse classi concrete che forniscono un'implementazione dell'algoritmo.
- **`MovimentoDiagonale`**: implementa il movimento diagonale.
- **`MovimentoOrizzontale`**: implementa il movimento orizzontale.
- **`MovimentoL`**; implementa il movimento ad L.
- …

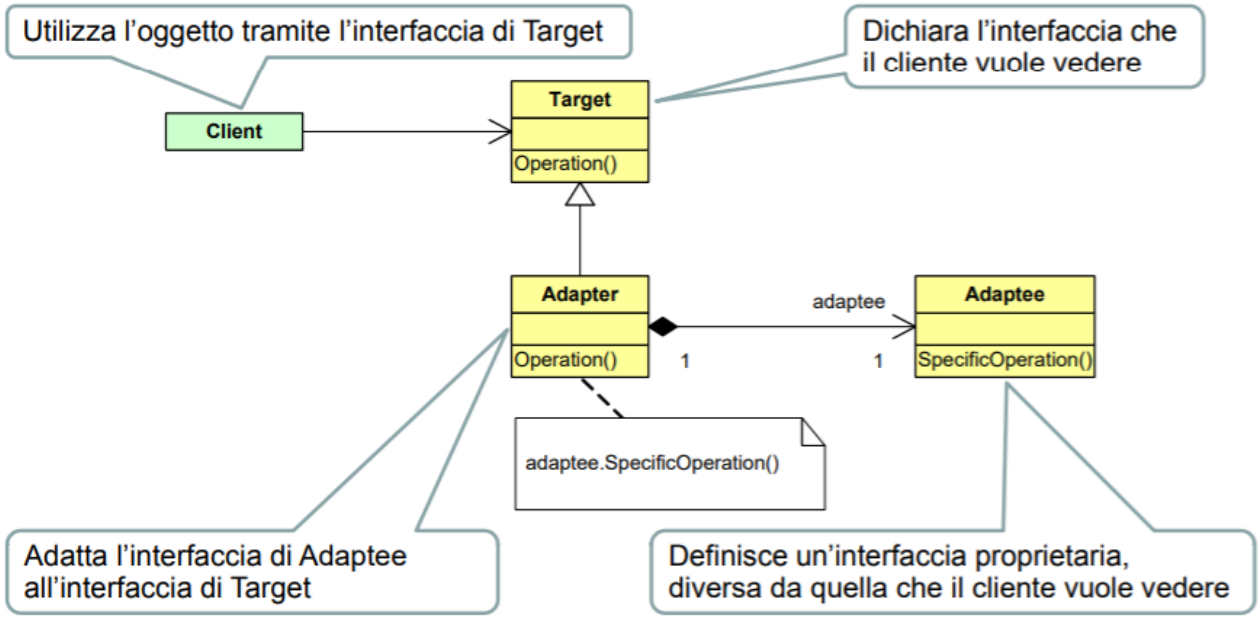
Pattern Adapter (strutturale)

Questo pattern converte l'interfaccia originale di una classe in un'interfaccia diversa che si aspetta il cliente.

Permette a classi che hanno interfacce incompatibili di lavorare insieme.

Si utilizza quando si vuole riutilizzare una classe esistente e la sua interfaccia non è conforme a quella desiderata. Questo pattern è noto anche come wrapper.

Si supponga che un **`Client`** dipenda da un'interfaccia **`Target`** già definita, la quale espone un certo metodo. Esiste una certa classe **`Adaptee`** che realizza tale metodo, ma ha un'interfaccia incompatibile con l'interfaccia **`Target`**. Di conseguenza, per risolvere questo problema, si utilizza una classe **`Adapter`**, che implementa **`Target`** e richiama il metodo di **`Adaptee`**, mascherandolo come una chiamata al metodo di **`Target`**.



Un esempio è il seguente:

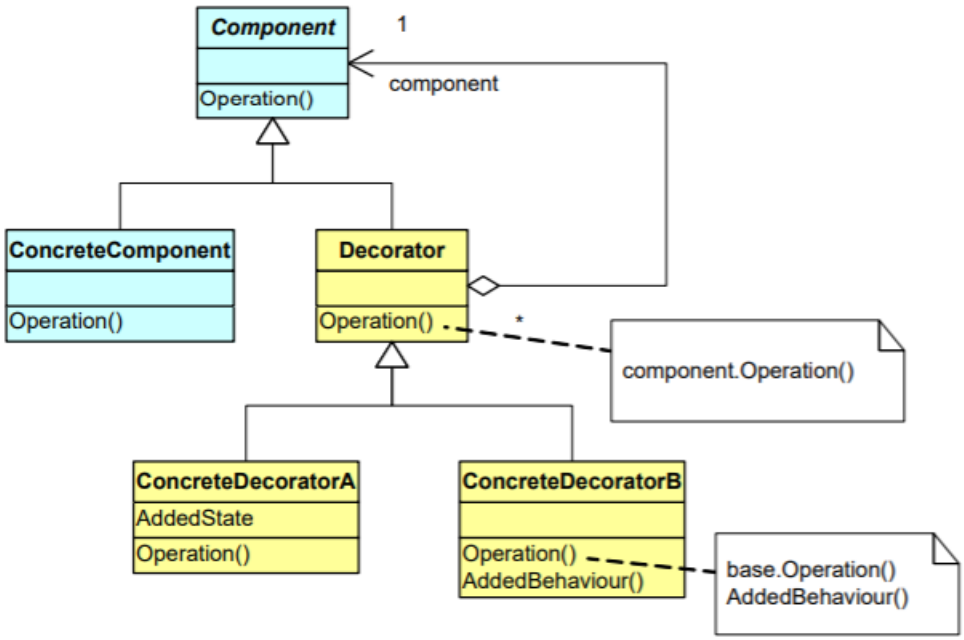
- Si ha un'interfaccia **`Visualizzatore`** che espone il metodo **`visualizzaImmagine()`**.
- Una classe **`VisualizzatoreJPG`** che permette di visualizzare un file JPG non compatibile con **`Visualizzatore`**.
- Il **`Client`** richiede a **`Visualizzatore`** di mostrare un'immagine JPG.
- Si realizza la classe **`VisualizzatoreAdapter`** che implementa **`Visualizzatore`** e mantiene un riferimento a **`VisualizzatoreJPG`**.

Quando il **`Client`** invoca il metodo **`visualizzaImmagine()`** di **`Visualizzatore`**, verrà invocato da **`VisualizzatoreAdapter`** che invoca il metodo di **`VisualizzatoreJPG`**, il tutto trasparentemente al **`Client`**.

Pattern Decorator (strutturale)

Questo pattern permette di aggiungere responsabilità a un oggetto dinamicamente. Fornisce un'alternativa flessibile all'ereditarietà quando si ha una gerarchia troppo estesa.

Il pattern decorator consente di aggiungere dinamicamente nuove funzionalità a un oggetto a run-time, mentre l'ereditarietà estende il comportamento delle classi padre alle classi figlie durante la fase di compilazione.

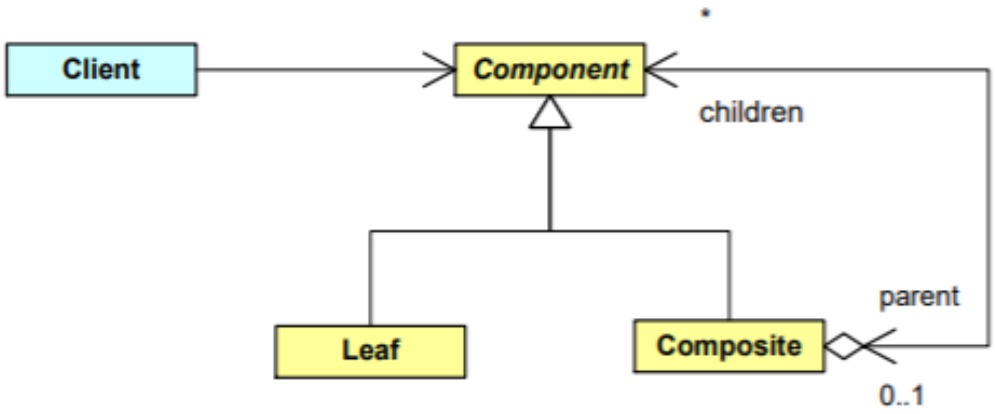


Un esempio di pattern decorator è il seguente:

- Si ha un'interfaccia **`IPanino`** che rappresenta **`Component`**.
- Si hanno le classi **`Hamburger`** e **`Hotdog`** che implementano **`IPanino`** e rappresentano i **`ConcreteComponent`**.
- Si vuole aggiungere altre proprietà dei quadrilateri, quindi si definisce la classe astratta **`PaninoExtra`** che rappresenta il **`Decorator`**.
- Si realizzano le classi che estendono **`PaninoExtra`**, come **`PaninoKetchup`**, **`PaninoMaionese`**, …, che rappresentano le **`ConcreteDecorator`**.

Pattern Composite (strutturale)

Questo pattern permette di comporre oggetti in una struttura ad albero al fine di rappresentare una gerarchia di oggetti contenitori – oggetti contenuti. Permette ai client di trattare in modo uniforme oggetti singoli e oggetti composti.



In questo pattern abbiamo:

- La classe astratta **`Component`** dichiara l'interfaccia e realizza il comportamento di default.
- Il **`Client`** accede e modifica gli oggetti della composizione attraverso l'interfaccia di **`Component`**.
- **`Leaf`** descrive il comportamento degli oggetti singoli che non possono avere figli.
- **`Composite`** definisce il comportamento degli oggetti aventi figli.

È necessario che il contenitore dei figli sia un attributo di **`Composite`**, e può essere di qualsiasi tipo (come array, lista, hashtable ecc). Il **`Client`** utilizza soltanto **`Component`**, pertanto quest'ultimo deve possedere tutti i metodi di cui il **`Client`** necessita e la definizione base dei metodi che dovranno essere ridefiniti dalle sottoclassi. Alcune operazioni definite in **`Component`** risultano prive di significato per gli oggetti senza figli, come **`add()`**, **`remove()`**, pertanto sono possibili due possibili approcci:

- Trasparenza: si definiscono le operazioni di **`add()`** e **`remove()`** in **`Component`**. Tuttavia, è possibile che il **`Client`** definisca operazioni illegali come l'aggiunta di figli alle foglie; per evitare questo problema è necessario che le foglie sollevino eventualmente un'eccezione per i metodi precedentemente citati.

- Sicurezza: i metodi di gestione dei figli come **`add()`** e **`remove()`** vengono implementati in **`Composite`**; occorre predisporre di un'implementazione che verifichi se l'oggetto che si sta trattando è un **`Composite`** o **`Leaf`**.

Supponiamo di avere:

- La classe **`Impiegato`** come **`Component`**.
- La classe **`Manager`** come **`Composite`**.
- La classe **`Programmatore`** come **`Leaf`**.

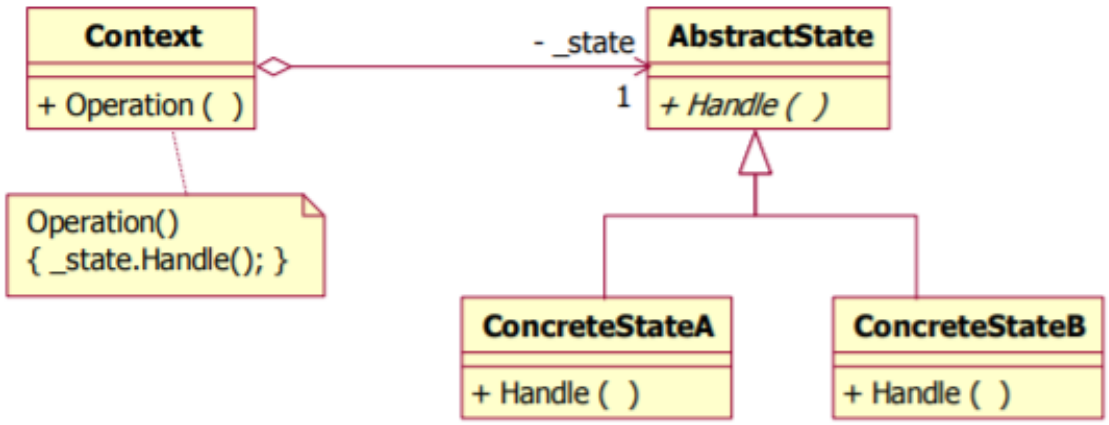
In questo caso il manager può avere sottoposti mentre (che siano altri manager o programmatori) il programmatore non può averne.

Quindi un'eventuale operazione **`aggiungiImpiegato()`** può essere eseguita da **`Manager`** ma non da **`Programmatore`**.

Pattern State (comportamentale)

Questo pattern serve a localizzare il comportamento specifico di uno stato e suddivide il comportamento in funzione dello stato.

Le classi concrete contengono la logica di transizione da uno stato all'altro, permette quindi di realizzare l'ereditarietà multipla.



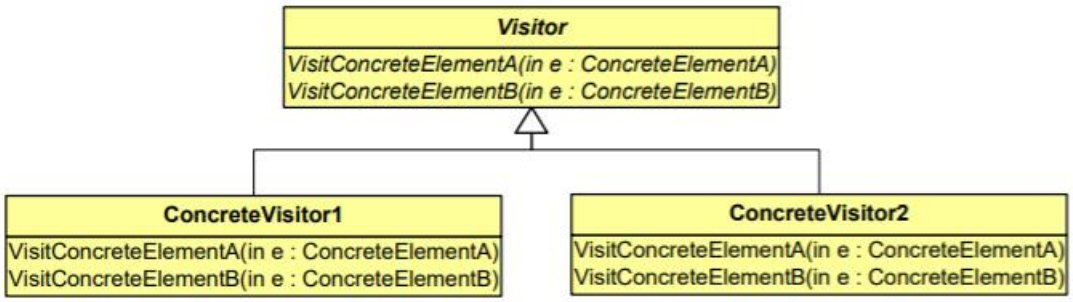
La classe astratta **`AbstarctState`** rappresenta lo stato. Questa classe viene estesa dai vari **`ConcreteState`** che sovrascrivono il metodo **`Handle()`**. Ad ogni cambiamento di stato la classe **`Context`** chiama **`Operation()`** che per implementare il comportamento che dipende dallo stato, a sua volta chiama **`Handle()`** del proprio stato corrente, il quale può a sua volta richiamare un metodo **`SetState()`** del cliente per cambiare lo stato.

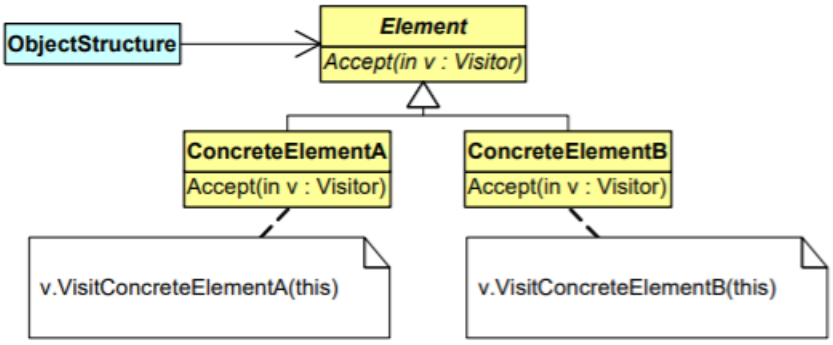
Un esempio può essere:

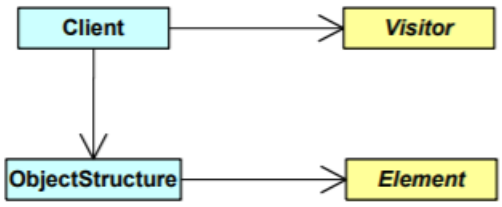
- Si ha la classe astratta **`TCPState`** che rappresenta **`AbstractState`**, con i metodi **`Open()`**, **`Close()`**, **`Ack()`**.
- La classi concrete **`ConcreteState`** sono rappresentate da **`TCPEstablished`**, **`TCPListen`**, **`TCPClosed`**, che ridefiniscono i metodi della classe astratta.
- Il contesto è rappresentato da **`TCPConnection`**.

Pattern Visitor (comportamentale)

Questo pattern permette di definire una nuova operazione da effettuare su gli elementi di una struttura, senza dover modificare le classi degli elementi coinvolti.







I componenti fondamentali di questo pattern sono:

- **`Visitor`**: classe astratta o interfaccia che dichiara il metodo **`Visit`**.
- **`ConcreteVisitor`**: classe concreta che estende o implementa Visitor e consente di percorrere la struttura de oggetti coinvolta.
- **`Element`**: classe astratta o interfaccia che dichiara il metodo **`Accept`**, che ha come parametro un oggetto **`Visitor`**.
- **`ConcreteElement`**: classe concreta che implementa o estende **`Element`**.
- **`ObjectStructure`**: realizzata come Composite o normale collezione, dichiara un'interfaccia che permette al **`Client`** di far visitare la struttura ad un **`Visitor`**.

Per definire una nuova operazione occorre implementare un nuovo **`ConcreteVisitor`**.

L'operazione che deve essere effettuata dipende da **`Visitor`** e da **`Element`**, quindi **`Accept`** è un'operazione di tipo double dispatch.

Un esempio può essere:

- Definiamo la classe astratta **`Prodotto`** che rappresenta **`Element`**, che presenta il metodo **`accept(Visitor v)`**.
- Le classi concrete che estendono **`Prodotto`** sono **`Pane`** e **`Pasta`**.
- Si definisce l'interfaccia **`Visitor`** che implementa i metodi **`visitPane(Pane pane)`** e **`visitPasta(Pasta pasta)`**.
- Si definisce **`VisitorNegozio`** che implementa **`Visitor`** e che racchiude la logica di vendita di pane e pasta **`visitPane(Pane pane)`** e **`visitPasta(Pasta pasta)`**.
- Per aggiungere una nuova funzionalità che consente ad esempio il rifornimento del magazzino di pasta e pane, sarà sufficiente creare una classe **`VisitorRifornimento`** che implementa **`Visitor`** e ridefinisce **`visitPane(Pane pane)`** e **`visitPasta(Pasta pasta)`**.

Modello Lock-Modify-Unlock, vantaggi e svantaggi

Il modello Lock-Modify-Unlock (LMU) viene utilizzato in alcuni sistemi di controllo delle versioni e consente ad una persona per volta di modificare file di un repository. Questo modello serve ad ovviare il seguente problema:

- UtenteA e UtenteB eseguono il check-out e lavorano sulle proprie copie;
- UtenteA esegue il check-in;
- UtenteB esegue il check-in dopo UtenteA, sovrascrivendo le modifiche;

Quindi le modifiche fatte da UtenteA andranno perse.

Con LMU, ogni volta che un utente voglia modificare un file della cartella di lavoro deve:

- Prima di tutto bloccarlo (lock);
- Effettuare le proprie modifiche (modify);
- Infine, sbloccarlo (unlock);

Questo modello, tuttavia, non è esente da problemi:

- Un utente può dimenticarsi di sbloccare il file, in questo modo nessun altro utente potrà fare modifiche. Si generano quindi notevoli ritardi nello sviluppo.
- La serializzazione non è sempre necessaria, infatti, due utenti potrebbero modificare due parti distinte del progetto senza conflitti. Viene ulteriormente rallentato il processo di sviluppo.
- È presente un falso sentimento di sicurezza da parte degli sviluppatori: le modifiche a due file distinti non sono necessariamente indipendenti. A causa delle loro dipendenze i due file, in caso di modifica non adatta, potrebbero non funzionare più.

Questi problemi, quindi portano a preferire questo modello solo nel caso di file che non possono essere facilmente uniti, come i file immagine.

Modello Copy-Modify-Merge, vantaggi e svantaggi

Il modello Copy-Modify-Merge (CMM) viene utilizzato in alcuni sistemi di controllo delle versioni. In questo modello è assente il meccanismo di blocco dei file.

In questo modello qualunque utente può eseguire l'accesso al repository del progetto e crea una copia locale e personale su cui lavorare. Al momento del check-in le modifiche effettuate dai vari utenti sono unite (non sovrascritte) fra di loro. Non è importante sapere chi sono gli altri utenti, ma solo le modifiche che hanno apportato.

Prima di eseguire il check-in, il sistema si accorge che le modifiche sono state effettuate su una versione precedente del file e pretende che l'utente aggiorni il proprio file con tutte le modifiche effettuate dagli altri utenti nel frattempo.

L'operazione che permette di unire le modifiche fatte dall'utente a quelle fatte dagli altri è chiamata merge. Il merge può avere due esiti:

- Successo: le modifiche effettuate non causano problemi di congruenza del codice.
- Conflitto: due o più utenti hanno modificato lo stesso blocco di codice generando incongruenze. In questo caso si risolvono i conflitti manualmente.

Anche in caso di conflitto, il tempo di risoluzione è sempre molto minore rispetto al ritardo causato da un lock nel modello Lock-Modify-Unlock.

Ovviamente il merge può andare a buon fine ma il programma non funzionare correttamente, questo perché il merge non rileva eventuali problemi logici, concettuali o semantici.

Modello a cascata e sue criticità

Il waterfall model è un modello di processo di sviluppo software che prevede fasi sequenziali ben distinte fra loro:

- Studio di fattibilità
- Analisi dei requisiti
- Analisi del problema
- Progettazione
- Implementazione
- Collaudo
- Manutenzione

Questo modello è fondato sul presupposto che modifiche sostanziali in fasi avanzate dello sviluppo richiedano un costo infattibile, quindi ogni fase deve essere svolta in maniera esaustiva prima di passare alla successiva (la retroazione porta a modifiche eccessive).

È importante definire:

- Semilavorati: sono i prodotti ottenuti in una fase ed utilizzati nella fase successiva. Sono costituiti da documentazione cartacea, codice dei singoli moduli e sistema nel suo complesso.
- Date: scadenza entro la quale devono essere prodotti i semilavorati, in modo da tener traccia dal progresso di lavoro.

Sono principalmente due i fattori che rendono il waterfall model efficiente:

- Immutabilità dell'analisi: i clienti definiscono tutti i requisiti che il software deve avere sin da subito, quindi nella fase iniziale del progetto si possono definire tutte le funzionalità che il software deve avere.
- Immutabilità del progetto: è possibile progettare l'intero sistema prima di aver scritto una sola riga di codice.

Come tutti i modelli, anche questo presenta degli svantaggi dovuti alla sua rigidità:

- Man mano che il sistema prende forma, le specifiche cambiano in continuazione ed anche la visione del cliente può cambiare (retroazione, modifiche costose).
- Spesso per avere migliori prestazioni occorre revisionare il progetto.

Per evitare questi problemi dovuti alla costosa retroazione si è deciso di limitare la retroazione ad un solo livello.

Per evitare problemi rispetto la visione complessiva del progetto (funzionalità) può essere l'ideale presentare al cliente un prototipo, cioè un modello approssimato dell'applicazione.

Il prototipo deve essere sviluppato in tempi molto brevi e con costi molto limitati. Una volta esaurito il compito, il prototipo viene abbandonato e si procede a costruire il sistema reale secondo i canoni del modello a cascata.

Questo approccio potrebbe risultare molto dispendioso da eliminare i vantaggi del modello a cascata.

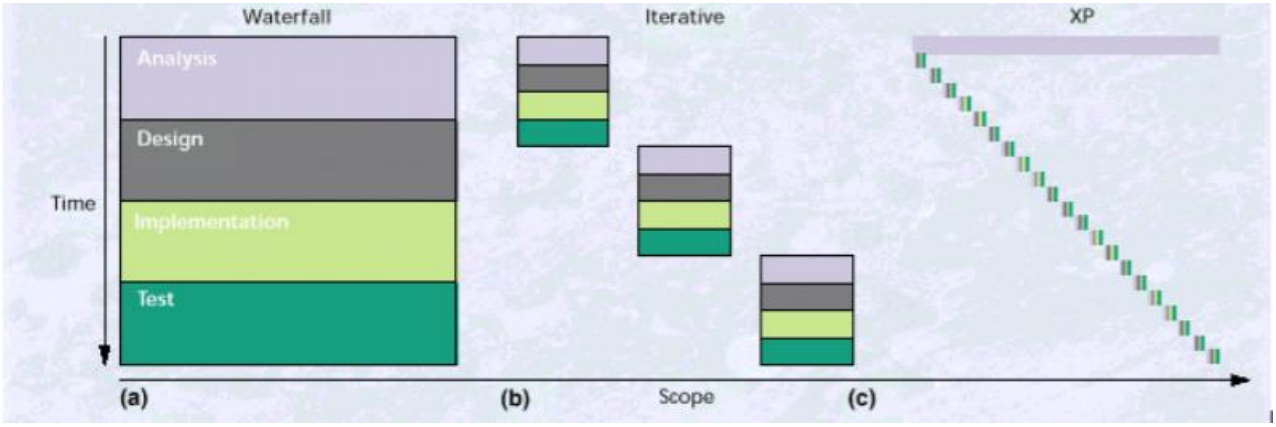
Modelli evolutivi XP

Esistono molti modelli di tipo evolutivo, tutti basati sull'idea di un ciclo di sviluppo in cui un prototipo iniziale evolve gradualmente verso il prototipo finito.

Il vantaggio fondamentale è che ad ogni interazione è possibile:

- Confrontarsi con gli utenti per quanto riguarda le specifiche e le funzionalità.
- Rivedere le scelte di progetto.

Questi modelli si sono orientati verso cicli sempre più brevi e interazioni sempre più veloci, fino ad arrivare al modello più radicale che prende il nome di Extreme Programming (XP).



Lo scopo di XP è di ridurre al minimo i costi in caso di cambiamenti nello sviluppo del software.

Questo modello è basato su quattro principi:

- Comunicazione: tra gli sviluppatori e tra gli sviluppatori e i clienti.
- Testing: il codice per i test deve essere molto grande, anche più di quello del programma vero e proprio.
- Semplicità: il codice deve essere estremamente semplice; non è conveniente fare codice più complesso cercando di prevedere sviluppi futuri. Il codice semplice inoltre è più riutilizzabile.
- Coraggio: non si deve esitare nelle modifiche al sistema. Il sistema deve essere ristrutturato ogni volta che si intravede un possibile miglioramento.

Sviluppo incrementale-iterativo

Per sistemi composti da sottosistemi, è possibile per ogni sottosistema adottare un diverso modello di sviluppo:

- Modello evolutivo: per sottosistemi con specifiche ad alto rischio.
- Modello a cascata: per sottosistemi con specifiche ben definite.

Di norma è conveniente creare e raffinare prototipi funzionanti dell'intero sistema o di sue parti secondo l'approccio incrementale-iterativo.

Con sviluppo incrementale:

- Costruiamo il sistema sviluppandone sistematicamente e in sequenza parti ben definite.
- Una volta costruita una parte questa non viene più modificata.
- È importante avere requisiti ben definiti della parte da progettare.

Con sviluppo iterativo:

- Si effettuano molti passi dell'intero ciclo di sviluppo del software, in modo da costruire iterativamente tutto il sistema, aumentandone ogni volta il livello di dettaglio.

Con sviluppo incrementale-iterativo:

- Si individuano le sotto-parti autonome;
- Si realizza il prototipo di una di esse;
- Si continua con le altre parti;
- Si aumenta progressivamente estensione e dettaglio dei prototipi;
- …

Rational Unified Process

Il Rational Unified Process (RUP) rappresenta un modello di processo software iterativo e ibrido, ed è pensato per il processo di sviluppo di software molto grandi.

RUP individua tre diverse visioni del processo di sviluppo:

- Prospettiva dinamica
- Prospettiva statica
- Prospettiva pratica

La prospettiva dinamica mostra l'evoluzione del modello nel tempo ed è composta da quattro fasi:

- Avvio: lo scopo è quello di delineare il business case, che può essere diviso in:
  - Tipo di mercato: individuare a quale settore mira il progetto e identificare gli elementi importanti affinché esso abbia successo commerciale.
  - Entità esterne: identificare tutte le persone o altri sistemi che interagiranno con il sistema.
  - Strumenti utilizzati: modello dei casi d'uso, valutazione dei rischi, definizione dei requisiti, …
- Elaborazione: comprende l'analisi del dominio e una prima fase di progettazione dell'architettura, in questa fase è necessario soddisfare alcuni requisiti:
  - Modello dei casi d'uso completo all'80%
  - Descrizione dell'architettura del sistema
  - Revisione del business case e dei rischi
  - Pianificazione del progetto
  - …
- Costruzione: in questa fase si progetta, programma e testa il sistema. Al termine di questa fase si dovrebbe avere un sistema software funzionante e ben documentato.
- Transizione: il sistema passa dall'ambiente di sviluppo a quello del cliente, questa fase necessita di operazioni come:
  - Beta testing
  - Training degli utenti
  - …

La prospettiva statica si concentra sulle attività di produzione del software chiamate workflow. RUP identifica sei workflow principali e tre di supporto, e tutti possono essere attivi in ogni stadio del processo.

È importante sottolineare che RUP è stato progettato insieme ad UML, quindi la descrizione dei workflow è orientata ai modelli associati ad UML.

I sei workflow principali sono:

- Modellazione delle attività aziendali
- Requisiti: si identificano gli attori, i casi d'uso e i rischi.
- Analisi e progetto: creazione e documentazione del progetto utilizzando modelli architetturali, dei componenti e sequenziali.
- Implementazione: viene scritto il codice relativo al progetto, sfruttando il più possibile la generazione automatica.
- Test: vengono effettuati test approfonditi su tutti i sottosistemi.
- Rilascio: viene creata una release del prodotto, distribuita agli utenti e installata.

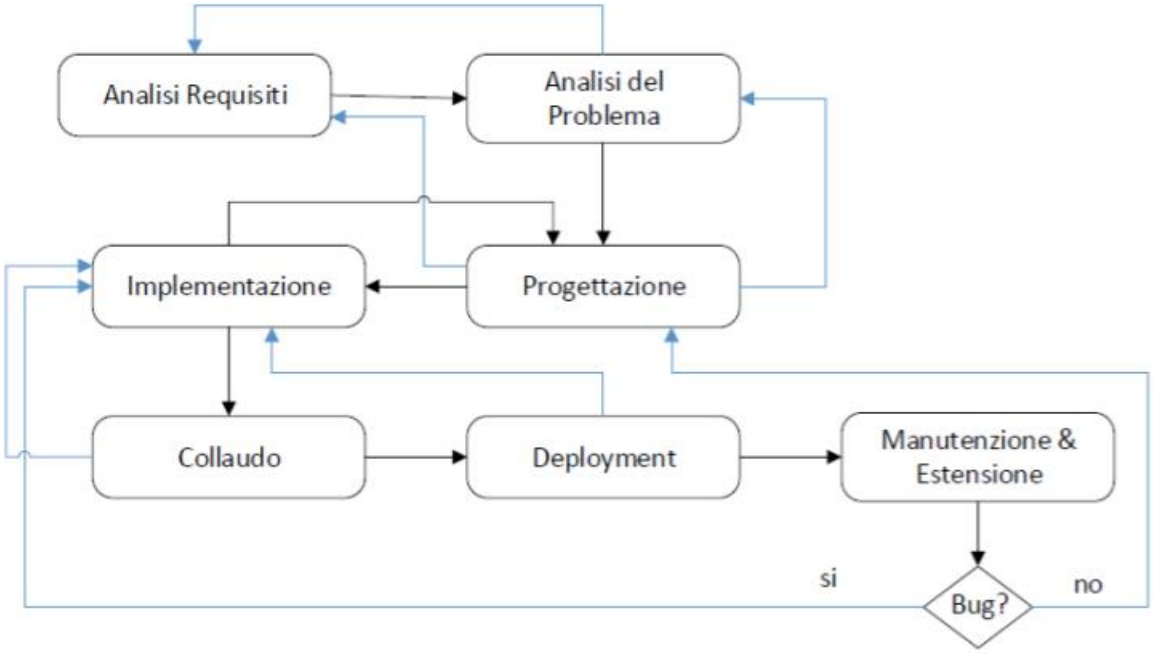
I tre workflow di supporto sono:

- Gestione della configurazione delle modifiche: per i cambiamenti
- Gestione del progetto: per lo sviluppo del sistema
- Ambiente: per rendere disponibili strumenti adeguati al team di sviluppatori

La prospettiva pratica di RUP descrive la buona prassi di ingegneria del software che si raccomanda di usare durante lo sviluppo di sistemi. Le pratiche fondamentali sono sei:

- Sviluppare software ciclicamente: si pianificano gli incrementi che saranno fatti sul sistema e si sviluppano e rilasciano prima le funzioni con priorità più alta.
- Gestire i requisiti: documentare tutti i requisiti del cliente e prima di apportare cambiamenti al sistema analizzare l'impatto che avranno sull'intero progetto.
- Usare architettura basate sui componenti
- Creare modelli visivi del software: sfruttare modelli grafici come UML per rappresentare visioni statiche e dinamiche.
- Verificare la qualità del software
- Controllare le modifiche al software

Processo di sviluppo simil-RUP:



Cosa sono i requisiti? Elencarne le tipologie.

I requisiti di un sistema rappresentano la descrizione dei servizi forniti e dei vincoli operativi. Il processo di ricerca, analisi, documentazione e verifica dei requisiti è chiamato ingegneria dei requisiti.

Una prima classificazione di requisiti può essere fatta in:

- Requisiti utente: in cui si dichiarano quali servizi il sistema deve offrire e i vincoli sotto i quali deve operare. Sono requisiti molto astratti ed espressi in linguaggio naturale, sono definiti in accordo ad una fase interlocutoria con il committente.
- Requisiti di sistema: definiscono le funzioni, i servizi e i vincoli operativi del sistema in modo dettagliato. È quindi una descrizione molto dettagliata su quello che il sistema dovrebbe fare. Il Documento dei Requisiti del Sistema deve essere preciso e definire esattamente cosa deve essere sviluppato.

I requisiti di sistema vengono solitamente suddivisi in:

- Requisiti funzionali: descrivono ciò che il sistema dovrebbe fare. Sono in pratica elenchi di servizi che il sistema dovrebbe fornire, o meglio come reagire a particolari input o come comportarsi in determinate situazioni.
- Requisiti non funzionali: non riguardano specifiche che si collegano direttamente alle funzionalità del prodotto. Questi requisiti possono essere molto difficili da verificare perché spesso sono molto vaghi e mescolati (contrasto o interazione) con requisiti funzionali. Sono divisi in:
    - Requisiti del prodotto: specificano o limitano le proprietà del sistema, in termini di affidabilità, prestazioni, protezione dati, tempo di risposta, …
    - Requisiti organizzativi: vincoli del processo di sviluppo, in termini di software da utilizzare, linguaggi per l'implementazione, …
    - Requisiti esterni: sono vincoli che derivano da fattori non provenienti dal sistema e dal suo processo di sviluppo, riguardano requisiti legislativi, etici, interoperabilità con sistemi esterni, …
- Requisiti di dominio: questi requisiti derivano dal dominio dell'applicazione. Includono una terminologia propria del dominio o riferisco ai suoi concetti. Sono requisiti fondamentali per costruire il dominio applicativo del prodotto.

Sistemi critici ed esempi di attacchi

I sistemi critici sono sistemi da cui dipendono persone o aziende. Se questi sistemi non erogano i loro servizi come e quando ci si aspetta possono verificarsi molti problemi e perdite economiche importanti. Possiamo classificare tre sistemi critici:

- Sistemi safety-critical: i fallimenti provocano incidenti, perdite umane o seri danni ambientali.
- Sistemi mission-critical: i fallimenti causano il fallimento di attività.
- Sistemi business-critical: i fallimenti portano a costi elevati per le aziende.

La proprietà più importante per un sistema critico è la sua fidatezza, che si misura come la somma di tre fattori:

- Disponibilità
- Affidabilità
- Sicurezza e protezione

I componenti di un sistema che possono causare fallimenti sono:

- Hardware: può fallire a causa di errori nella progettazione, guasti a un componente o per usura.
- Software: può fallire a causa di errori nelle specifiche, nella progettazione o implementazione.
- Operatori umani: dipendente distratto.

Con l'aumentare dell'affidabilità di software e hardware gli errori umani sono diventati la causa più probabile di fallimenti del sistema.

La sicurezza e la protezione dei sistemi critici sono diventati sempre più importanti con l'aumentare delle connessioni di rete (sistemi più vulnerabili, ma patch più facilmente reperibili).

Esempi di attacchi possono essere:

- BufferOverflow
- Sniffing
- Spoofing
- Hijacking
- Dos
- Ddos
- Trojan

Ciclo di vita della valutazione del rischio

L'analisi del rischio si occupa di bilanciare eventuali perdite, dovute ad attacchi informatici, con i costi richiesti per assicurare la protezione dei beni. Un'importante componente dell'analisi del rischio è la valutazione del rischio, composta da più fasi:

- Valutazione preliminare del rischio: determina i requisiti di sicurezza dell'intero sistema.
- Ciclo di vita della valutazione del rischio: avviene parallelamente al ciclo di vita dello sviluppo del software. In questa fase occorre conoscere l'architettura del sistema e l'organizzazione dei dati. La scelta della piattaforma e del middleware è stata già effettuata, così come la strategia di sviluppo del sistema; ciò consente di conoscere meglio cosa è necessario proteggere e quali sono le possibili vulnerabilità del sistema, alcune delle quali determinate da scelte progettuali precedenti. In questa fase vengono effettuate l'identificazione e la valutazione della vulnerabilità, ovvero quali beni hanno la maggiore probabilità di essere colpiti. Il risultato della valutazione del rischio è un insieme di decisioni ingegneristiche che influenzano la progettazione o l'implementazione del sistema e limitano il suo utilizzo.

Categorie requisiti di sicurezza

Non è sempre possibile specificare i requisiti associati alla sicurezza in modo quantitativo. Molto spesso questa tipologia di requisiti può essere espressa nella forma “non deve”, cioè vengono definiti comportamenti inaccettabili del sistema. I requisiti di sicurezza specificano il contesto, i beni da proteggere e il valore che questi ultimi hanno per l’organizzazione. Le categorie dei requisiti per la sicurezza sono:

- Requisiti di identificazione: specificano se il sistema deve identificare gli utenti prima dell’interazione con essi.
- Requisito di autenticazione: specifica come identificare gli utenti.
- Requisiti di autorizzazione: specificano privilegi e permessi di accesso per utenti autenticati.
- Requisiti di immunità: specificano come il sistema deve proteggersi da virus, worm e minacce simili.
- Requisiti di integrità: specificano come evitare la corruzione dei dati.
- Requisiti di scoperta delle intrusioni: specificano meccanismi di rilevamento degli attacchi.
- Requisiti di non-ripudiazione: specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento.
- Requisiti di riservatezza: specificano come mantenere le informazioni riservate.
- Requisiti di controllo della protezione: specificano come il sistema può essere controllato.
- Requisiti di protezione della manutenzione del sistema: specificano come un’applicazione può evitare modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione.

Linee guida di progettazione della sicurezza

Non esisto regole rigide e ben definite per ottenere un sistema sicuro. Sistemi differenti richiedono tecniche differenti per ottenere un adeguato livello di sicurezza. Tuttavia, ci sono linee guida per la progettazione di sistemi sicuri che servono come:

- Mezzo per migliorare la consapevolezza dei problemi legati alla sicurezza al team di progettisti software.
- Base per una lista di controlli da fare in fase di test del sistema.

Possiamo articolare queste linee guida in 10 punti:

- Basare le decisioni della sicurezza su una politica esplicita: le decisioni della sicurezza vengono presentate in un documento di alto livello che definisce come ottenere la sicurezza. I progettisti devono consultare questo documento sia nelle decisioni della progettazione che nella loro valutazione.
- Evitare un singolo punto di fallimento: è buona norma in sistemi critici evitare un singolo punto di fallimento, questo perché un singolo fallimento in una parte del sistema non deve trasformarsi nel fallimento di tutto il sistema. Questo significa che non ci si dovrebbe affidare ad un singolo meccanismo di sicurezza, ma si dovrebbero impiegare differenti tecniche. (es. autenticazione a due fattori).
- Fallire in un certo modo: supponendo che il fallimento del sistema sia inevitabile, un sistema critico dovrebbe fallire in modo sicuro. Quindi anche se il sistema fallisce non deve essere consentito ad un attaccante di accedere ai dati.
- Bilanciare sicurezza e usabilità: sicurezza e usabilità tipicamente sono in forte contrasto, infatti, ogni volta che si aggiunge una caratteristica di sicurezza il sistema inevitabilmente diventa meno usabile. (es. invece di introdurre un'autenticazione a due fattori obbligare l'utente ad inserire password robuste).
- Essere consapevoli dell'esistenza dell'ingegneria sociale: attraverso l'ingegneria sociale un attaccante potrebbe ingannare un utente che può interagire con il sistema (dipendente, utente con credenziali valide, …) in modo da rivelare informazioni riservate. Dal punto di vista progettuale contrastare l'ingegneria sociale è quasi impossibile. Meccanismi di log possono tener traccia di anomalie.
- Usare ridondanza e diversità riduce i rischi: con ridondanza si intende mantenere più versioni del software e dei dati del sistema, mentre diversità

significa che le diverse versioni del sistema non dovrebbero usare la stessa piattaforma o essere basati sulle stesse tecnologie. In questo modo una vulnerabilità su una versione non influenzerà le altre.

- Validare gli input: è fondamentale validare gli input inseriti da parte degli utenti, per evitare attacchi di injection (SQL injection, …) o buffer overflow.
- Dividere in compartimenti i beni: l'utente deve poter accedere solo alle informazioni strettamente necessarie (principio del minimo privilegio). In questo modo eventuali attacchi sono molto più contenuti.
- Progettare per il deployment: molti problemi di sicurezza sorgono perché il sistema non viene configurato correttamente (misconfiguration).
- Progettare per il ripristino: ogni sistema deve essere progettato con il presupposto che possano avvenire errori legati alla sicurezza. Si deve quindi pensare a come ripristinare eventualmente il sistema in uno stato operazionalmente sicuro.

Linee guida progettazione per il deployment

La configurazione e il deployment sono spesso visti come problemi di amministrazione di sistemi, quindi fuori da processo di progettazione.

Tuttavia, i progettisti software hanno la responsabilità di progettare per il deployment, in modo da fornire supporti che riducano la probabilità che gli amministratori incontrino errori in fase di configurazione.

Le linee guida per la progettazione per il deployment sono le seguenti:

- Includere supporto per visionare e analizzare le configurazioni: includere sempre programmi di utilità per consentire agli amministratori di esaminare la configurazione del sistema, idealmente evidenziando le impostazioni critiche per la sicurezza.
- Minimizzare i privilegi di default: il software deve essere progettato in modo che la configurazione di default garantisca il minimo privilegio possibile per il corretto funzionamento del programma. Questo per limitare i danni di un possibile furto di credenziali, privilege escalation, …
- Localizzare le impostazioni di configurazione: file di configurazione che appartengono alle risorse di una stessa parte del sistema devono essere nella stessa posizione.
- Fornire modi per rimediare a vulnerabilità di sicurezza: sono necessari meccanismi di aggiornamento del sistema e riparazione circa le vulnerabilità di sicurezza. Includere quindi verifiche automatiche per aggiornamenti di sicurezza e download di tali aggiornamenti appena disponibili.

Testare la sicurezza, black box – white box

I test giocano un ruolo fondamentale nel processo di sviluppo di un software. L'area relativa alla sicurezza dovrebbe essere fortemente testata, questi tipi di test sono generalmente più lunghi e tediosi dei test funzionali.

Possiamo dire che ci sono due categorie di test riguardanti la sicurezza:

- Test tradizionali, per accertare la sicurezza dei requisiti applicativi, e sono condotti tipicamente dal team di testing.
- Test di rottura, condotti da esperti della sicurezza.

I test di rottura sono divisi in:

- Black Box testing: questo test assume come base la non conoscenza dell'applicazione. I tester affrontano l'applicazione come farebbe un vero attaccante, quindi indagando su informazioni circa la struttura interna e tentando delle violazioni. Ci sono numerosi tool che permettono di scansionare, enumerare e indagare. In questo test non si considerano solo le debolezze legate al codice ma anche debolezze architetturali (errori di configurazione, problemi legati ai linguaggi, …).
- White box testing: questo test assume come base la completa conoscenza dell'applicazione, a partire dal codice sorgente. I tester operano una revisione del codice cercando eventuali falle legate alla sicurezza. I tool utilizzati sono diversi rispetto ai test black box, in questo caso si trattano di tool di debugging. In questo modo è possibile scoprire errori come:
    - Mancanza di verifica degli input
    - Problemi di corsa critici
    - Memory leak
    - …

Capacità di sopravvivenza del sistema

Per capacità di sopravvivenza del sistema si intende la capacità del sistema di continuare a fornire i servizi essenziali agli utenti legittimi in fasi critiche, cioè quando:

- Il sistema è sotto attacco;
- Parti del sistema sono danneggiate a seguito di un attacco o di un fallimento;

La capacità di sopravvivenza è una proprietà dell'intero sistema e non dei singoli componenti.

La disponibilità è quindi l'aspetto fondamentale della sopravvivenza e questa va studiata durante la progettazione del sistema.

Occorre conoscere:

- Quali sono i servizi critici e come possono essere compromessi
- La qualità minima dei servizi che deve essere mantenuta
- Come proteggere i servizi e come ripristinarli in caso di indisponibilità

È stato ideato il Survivable Analysis System per permettere di:

- Valutare le vulnerabilità del sistema
- Supportare la progettazione del sistema in modo da promuovere la propria sopravvivenza

Grazie a questo metodo la sopravvivenza del sistema dipende da tre strategie e quattro fasi.

Le strategie sono:

- Resistenza: resistere ai problemi attuando all'interno del sistema strategie per respingere attacchi, come le firme digitali.
- Identificazione: individuare problemi, quindi riconoscere attacchi e fallimenti, come checksum sui dati critici.
- Ripristino: ripristinare le complete funzionalità del sistema a seguito di un attacco o fallimento.

Le fasi invece sono:

1- Capire il sistema: riesaminare il sistema, i suoi obiettivi, i suoi requisiti e la sua architettura.
2- Identificare servizi critici: identificare quali servizi devono essere mantenuti costanti e quali compiti devono svolgere.

3- Simulare attacchi: identificare scenari o casi d'uso di possibili attacchi ai componenti del sistema.
4- Analizzare la sopravvivenza: identificare i componenti sia essenziali che a rischi e le strategie di sopravvivenza basate su resistenza, identificazione e ripristino.

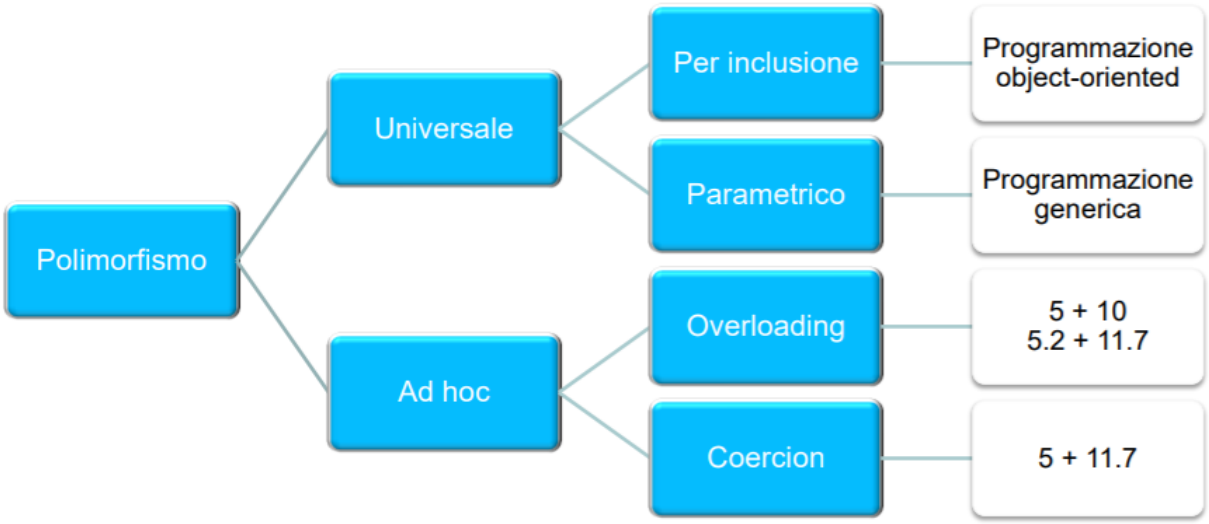
Sfortunatamente l'analisi della sopravvivenza non viene effettuata nella maggior parte dei processi di ingegnerizzazione, in quanto molte aziende che non hanno subito attacchi risultano scettiche nell'investire sulla sicurezza. È tuttavia consigliato quest'ultimo investimento, prevenendo eventuali attacchi, piuttosto che subirli, in quanto le perdite conseguenti risulterebbero gravi in termini di risorse e, nei casi peggiori, di vite.

Polimorfismo secondo Cardelli-Wegner

Il polimorfismo è la capacità di uno stesso elemento di assumere forme diverse in contesti diversi, o di elementi diversi di assumere la stessa forma in un determinato contesto.

La classificazione Cardelli-Wegner divide il polimorfismo in due macrocategorie:

- Universale: in questo caso gli elementi possono assumere infinite forme. Questa forma di polimorfismo è a sua volta divisa in:
  - Per inclusione: è quello utilizzato nella programmazione orientata agli oggetti e utilizza:
    - Overriding: consente la ridefinizione di un metodo (definito nella super-classe) nella sottoclasse. Approccio decisamente più sicuro se il metodo è astratto.
    - Binding dinamico: consentito grazie alla VMT (Virtual Method Table) posseduta da ogni classe, e contiene un puntatore a tutti i metodi della classe.
  - Parametrico: è utilizzato nella programmazione generica rispetto ai tipi. Consiste nel definire una classe in cui il tipo di una o più variabili è un parametro della classe stessa. Da ogni classe generica si generano classi indipendenti che non possiedono alcun rapporto di ereditarietà.
- Ad-hoc: in questo caso gli elementi assumono un numero finito di forme. Possiamo suddividerlo in:
  - Overloading: consente di ridefinire metodi e operatori e avviene in fase di programmazione.
  - Coercion: è una conversione implicita del tipo di una variabile.



Un esempio di polimorfismo per inclusione (in codice Java) può essere il seguente:

```
public class A{
     public void function(int x){}
     …
}
public class B extends A{
     @Override
     public void function(int x){…}
}
```

Il binding dinamico viene utilizzato per discriminare il metodo richiamato; la Virtual Method Table di ciascuna classe punta al metodo evocato.

Compilazione ed esecuzione del codice in .NET

Il Common Language Runtime (CLR) viene utilizzato in .NET come ambiente virtuale di esecuzione delle applicazioni. Il codice che viene eseguito in CLR prende il nome di codice gestito. Il codice sorgente viene trasformato dal compilatore .NET in codice IL (CLI assembly, ovvero .exe o .dll); un assembly contiene, oltre al codice IL, anche un manifest che fornisce informazioni come i tipi di assembly, la versione, e requisiti di sicurezza.

Il codice IL a sua volta viene convertito dal compilatore Just In Time (JIT) in codice nativo, che può essere eseguito.

Il CLR prevede servizi aggiuntivi come:

- Garbage collector: si occupa del ciclo di vita degli oggetti; qualora un oggetto non risulti più essere referenziato viene distrutto. A differenza di Component Object Model (COM), non viene considerato il reference counting, ovvero il conteggio dei riferimenti a ciascun oggetto; in questo modo si ha una velocità di allocazione maggiore. Sono inoltre consentiti i riferimenti circolari. Con questo approccio, tuttavia, si verifica la perdita della distruzione deterministica, ovvero una richiesta esplicita di liberazione della memoria occupata da un oggetto. Tramite garbage collector la memoria viene liberata in modo non deterministico, ovvero quando l'oggetto non risulta più raggiungibile.
- I/O su file;
- Gestione delle eccezioni: le eccezioni sono oggetti che ereditano dalla classe System.Exception. È possibile gestire le eccezioni sfruttando i seguenti tre concetti:
    - throw: lancio di un'eccezione;
    - catch: cattura di un'eccezione;
    - finally: esecuzione di codice di uscita da un blocco controllato.

Garbage collector in C#

Il Garbage Collector si occupa del rilascio delle risorse qualora esse non vengano più utilizzate da un oggetto. Consente di avere maggiore stabilità del programma, poiché evita che un programmatore manipoli direttamente puntatori ad aree di memoria.

Il vantaggio dell'utilizzo di un garbage collector è il contrasto delle seguenti problematiche:

- Dangling pointer: puntatori ad aree di memoria deallocate precedentemente, che potrebbero essere state successivamente assegnate a un altro oggetto.
- Doppia deallocazione: causata da più chiamate consecutive di deallocazione della stessa area di memoria.
- Memory leak: un oggetto non più utilizzato non viene deallocato, pertanto continua a occupare memoria.

Gli svantaggi invece sono:

- Vengono richieste maggiori risorse di calcolo;
- Incertezza del momento in cui viene effettuata la garbage collection;
- Il rilascio della memoria è non deterministico, ovvero non si sa il momento esatto in cui il rilascio avviene, né l'ordine di rilascio delle aree non più utilizzate. Ciò può dipendere dall'algoritmo utilizzato dal garbage collector.

L'ambiente .NET sfrutta come strategia di garbage collection il tracing, cioè si stabiliscono quali oggetti sono raggiungibili e si eliminano quelli non raggiungibili. Quando un processo viene inizializzato il Common Language Runtime (CLR) riserva una regione contigua di spazio di indirizzamento, noto come managed heap e memorizza l'indirizzo di partenza della regione in un puntatore chiamato **`NextObjPtr`**. Nel caso in cui venga eseguita una **`newobj`**, il CLR:

1- Determina la dimensione in byte dell'oggetto e aggiunge a quest'ultimo due campi che possono essere da 32 o 64 bit. Il primo campo contiene un puntatore alla tabella dei metodi, mentre il secondo è un campo **`SyncBlockIndex`**.
2- Controlla se a partire da **`NextObjPtr`** ci sia spazio sufficiente; in caso negativo viene utilizzato il Garbage Collector, o viene lanciata **`OutOfMemoryException`**.

3- Aggiorna i puntatori relativi all'oggetto appena creato e allo spazio libero di memoria:
```
thisObjPtr = NextObjPtr;
NextObjPtr += sizeof(oggetto);
```
4- Invoca il costruttore dell'oggetto.
5- Restituisce il riferimento all'oggetto.

Lo scopo del garbage collector è di individuare quali oggetti non vengono più utilizzati dall'applicazione; quest'ultima presenta un insieme di radici (root). Ciascuna radice è un puntatore a un oggetto di tipo riferimento sicuramente attivo, come:

- Variabili globali e field statici di tipo riferimento;
- Variabili locali o argomenti attuali di tipo riferimento presenti sugli stack dei vari thread;
- Registri della CPU contenenti gli indirizzi di oggetti di tipo riferimento.

Vengono distinte di tipologie di oggetti: gli oggetti vivi sono raggiungibili (direttamente o indirettamente) dalle radici, mentre gli oggetti garbage non lo sono.

Inizialmente il garbage collector marca tutti gli oggetti sul managed heap come garbage; successivamente viene interpellata la tabella delle radici, per stabilire quali oggetti siano vivi, marcandoli come tali.

Terminata la classificazione degli oggetti viene liberata la memoria occupata dagli oggetti garbage; tuttavia, ciò causa frammentazione nel managed heap poiché si hanno aree di memoria libere non contigue. Quindi si effettua una compattazione della memoria in uso, modificando di conseguenza i riferimenti agli oggetti spostati;

Al termine dell'unificazione si aggiorna il valore di **`NextObjPtr`**.

Il Garbage collector può effettuare tutte le operazioni elencate in precedenza, in quanto conosce il tipo di un oggetto e può sfruttare i metadati per determinare quali campi dell'oggetto contengono riferimenti agli altri oggetti.

Nel caso in cui un oggetto contenga riferimenti a risorse di tipo unmanaged (file, connessione al database, socket, e così via) è responsabilità del programmatore occuparsi della fase di finalizzazione, ovvero il rilascio della risorsa prima della deallocazione.

Passaggio dei parametri in C#

Il passaggio dei parametri a un metodo può avere risultati distinti, in base alla tipologia di oggetto passato come argomento. Esistono tre tipologie di argomenti:

- In
    - Tipi valore: si ha passaggio per copia dell'oggetto e la modifica da parte del metodo invocato avviene solo sulla copia.
    - Tipi riferimento: si ha passaggio per copia del riferimento dell'oggetto, e la modifica da parte del metodo invocato avviene sulla copia del riferimento, ma non sul riferimento originale.

  In entrambi i casi gli argomenti passati ai metodi devono essere stati già inizializzati.
- In/Out
    - Tipi valore: si ha passaggio per riferimento e le modifiche influenzano l'oggetto originale
    - Tipi riferimento: si ha passaggio per riferimento dell'indirizzo dell'oggetto e le modifiche influenzano l'oggetto referenziato, ma non l'oggetto originale

  In entrambi i casi gli argomenti passati ai metodi devono essere stati già inizializzati.
- Out: si ha passaggio, sia per gli oggetti di tipo riferimento che di tipo valore, dei loro indirizzi, e le modifiche hanno effetto sul chiamante. L'inizializzazione degli oggetti deve avvenire nel metodo in cui sono stati passati come argomenti.

Delegato in C#

I delegati sono oggetti che contengono un riferimento a un metodo da invocare. Sono equivalenti ai puntatori a funzioni (functor) presenti nei linguaggi C/C++, ma orientati agli oggetti ed efficaci.

Eseguono funzionalità di callback, tra le quali:

- Elaborazione asincrona: permette a un metodo di accettare un delegato come parametro e di chiamare il delegato in un momento successivo. Questo tipo di elaborazione viene eseguita quando un processo impiega molto tempo ad essere completato; quando viene terminato il chiamante viene avvertito.
- Elaborazione cooperativa: viene fornita una parte del servizio dal metodo chiamato e la parte rimanente viene effettuata dal chiamante. Un esempio di elaborazione cooperativa è il merge sort.
- Gestione degli eventi: ogni entità interessata a un determinato evento si registra presso il generatore dell'evento, specificando quale metodo gestirà l'evento.

L'istanza di un delegato può incapsulare uno o più metodi, specificando gli argomenti di ciascun metodo e il valore restituito. Ciascun metodo è riferito a un'entità richiamabile che può essere:

- Un metodo, nel caso di metodi statici;
- Un'istanza e il metodo relativo a quell'istanza, nel caso di istanze di metodi.

Un delegato contiene soltanto la firma del metodo, e non conosce la classe o il metodo a cui si sta riferendo; è quindi ideale per l'invocazione anonima.

Nel caso in cui si abbia l'invocazione di un'istanza delegata, che possiede una o più voci nell'elenco di invocazione, si invocano i metodi dell'elenco in ordine e in modo sincrono.

Il delegato è inoltre un'entità type-safe che si pone tra un chiamante e zero o più call target e che si comporta come un'interfaccia con un solo metodo. Per ciascun metodo invocato vengono passati come argomenti gli stessi forniti all'istanza del delegato. Si hanno due scenari:

- Nel caso in cui vengono inclusi parametri di riferimento nell'invocazione del delegato, ciascuna invocazione del metodo avviene con un riferimento alla medesima variabile e le modifiche alla variabile da parte di un metodo nell'elenco di invocazione saranno visibili ai metodi successivi nell'elenco di invocazione.

- Nel caso di invocazione del delegato con parametri di output o valori di ritorno il loro valore finale sarà determinato dall'invocazione dell'ultimo delegato presente nell'elenco.

Evento in C#

Un evento viene utilizzato al posto di un delegato in quanto quest'ultimo utilizza campi pubblici per la registrazione di metodi.

Event consente di rendere automatico il supporto per la pubblica registrazione e l'implementazione privata. Si possono fornire gestori di registrazione eventi definiti dall'utente: un possibile vantaggio di questo approccio è il controllo.

Un evento viene scaturito dall'interazione con l'utente o in seguito alla logica del programma. Esistono tre entità relative agli eventi:

- Event sender: la sorgente dell'evento è l'oggetto o la classe che scaturisce l'evento.
- Event receiver: l'oggetto o la classe per il quale l'evento è determinante e quindi desidera di venire notificato in caso di verifica dell'evento.
- Event handler: il metodo dell'event receiver, eseguito nel momento in cui avviene la notifica.

Quando un evento si verifica, il sender invia un messaggio di notifica a tutti i receiver; in genere il sender non ha modo di conoscere né gli handler, né i receiver e grazie all'utilizzo di delegati si collegano sender e receiver, consentendo invocazioni anonime.

L'evento incapsula il delegato, pertanto occorre dichiarare un tipo di delegato prima di dichiarare un evento. I delegati possiedono due parametri:

- La fonte che ha scatenato l'evento;
- Dati dedicati all'evento.

Molti eventi non possiedono dati, pertanto è sufficiente utilizzare il delegato presente nella classe **`System.EventHandler`**. Occorre definire nuovi delegati per gestire gli eventi soltanto quando questi ultimi possiedono dei dati. Nel caso in cui un evento non debba passare ulteriori informazioni ai propri gestori viene sfruttata la classe **`System.EventArgs`**, aggiungendo i dati necessari e utilizzando il delegato **`EventHandler<TEventArgs>`**.

Nel caso in cui si dichiari un evento, come **`Public event EventHandler changed`**, La keyword event ne limita sia le possibilità di utilizzo che la visibilità; successivamente l'evento viene trattato come un delegato di tipo speciale, e può diventare null se nessun cliente si è registrato; può venire associato a uno o più metodi da invocare.

Dato che un evento può essere visto come un delegato, sono consentite soltanto alcune operazioni del delegato, come l'aggiunta di un delegato

all’evento (grazie all’operatore +=) oppure sganciarsi da un evento (mediante l’operatore -=).

Per ricevere notifiche relative a quell’evento occorre che il cliente definisca un metodo **`EventHandler`**, che verrà evocato all’atto della notifica dell’evento; dopodiché si crea un delegato dello stesso tipo dell’evento, viene riferito al metodo e successivamente lo si aggiunge alla lista dei delegati di quell’evento. Grazie alle singole operazioni += e -= ammesse nessuno può ottenere o modificare l’elenco dei metodi sottostante dei gestori di eventi.

Metaprogrammazione e riflessione in C#

La metaprogrammazione viene utilizzata per programmare un sistema in modo che abbia accesso alle informazioni relative al sistema stesso, e che possa manipolare tali informazioni. Essa viene realizzata mediante la riflessione, implementata in C# tramite la classe `System.Reflection`.

La riflessione sfrutta i metadati, ovvero dati che descrivono altri dati; infatti, se un componente dispone di informazioni sufficienti per descrivere se stesso, le sue interfacce possono essere esplorate dinamicamente. In Java e .NET i dati sono generati nel momento in cui si definisce un tipo, vengono salvati assieme alla sua definizione e sono disponibili a runtime; in COM e CORBA i metadati sono definiti da Interface Definition Language (IDL).

La riflessione viene utilizzata per:

- Esaminare i dettagli di un assembly;
- Istanziare oggetti e chiamare metodi scoperti a runtime;
- Creare, compilare ed eseguire assembly, ove necessario.

La chiave per la riflessione in .NET è la classe `System.Type`:

- Tutti gli oggetti sono istanze di tipi, e i tipi stessi sono istanze di `System.Type`;
- Il tipo di un oggetto può essere scoperto tramite il metodo `GetType()`;
- È presente un solo oggetto `Type` per ogni tipo definito.

I membri pubblici sono sempre accessibili, mentre i membri non pubblici lo sono solo se il chiamante ha permessi sufficienti. Per creare dinamicamente istanze di oggetti si usa la classe `System.Activator`, il cui metodo `CreateInstance(Type type, Object[] args)` è equivalente all'operazione `new`. Per aggiungere informazioni ai metadati è possibile utilizzare attributi personalizzati, ovvero classi visibili via riflessione che derivano da `System.Attribute` e che possono contenere proprietà e metodi.

Per creare attributi personalizzati occorre:

- Dichiarare la classe dell'attributo;
- Dichiarare i costruttori;
- Dichiarare le proprietà;
- Opzionalmente applicare `AttributeUsageAttribute`, che specifica alcune caratteristiche della classe, ovvero a quali elementi l'attributo è applicabile, quando l'attributo può essere ereditato e quando possono esistere molteplici istanze di un attributo.

Il metodo **`GetCustomAttributes()`** restituisce la lista degli attributi personalizzati. La classe **`System.Reflection.Emit`** consente di scrivere il codice IL necessario per creare e compilare un assembly che potrà essere chiamato direttamente dal programma che lo ha creato, e che potrà essere archiviato su disco in modo che altri programmi possano utilizzarlo.

Ciclo di vita oggetto il linguaggio OO

In un ambiente object-oriented, ogni oggetto che deve essere utilizzato dal programma è descritto da un tipo e ha bisogno di un'area di memoria dove memorizzare il suo stato. I passi per utilizzare un oggetto di tipo riferimento sono:

1- Allocare la memoria per l'oggetto;
2- Inizializzare la memoria per rendere utilizzabile l'oggetto;
3- Usare l'oggetto;
4- Eseguire un clean up dello stato dell'oggetto, se necessario;
5- Liberare la memoria.

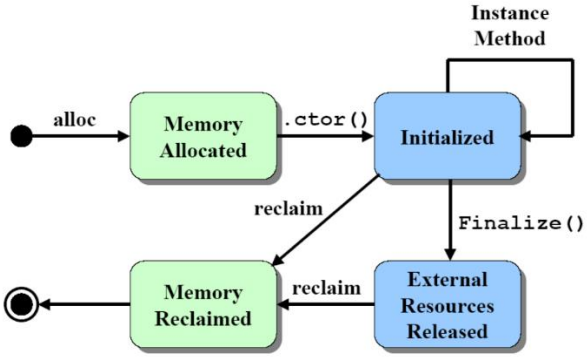
Per allocare la memoria ogni linguaggio ha le sue modalità: ad esempio in C si usano le funzioni malloc, calloc e realloc, in Java si usa new e in C# si usa newobj.

Quando si inizializza la memoria, a ogni variabile deve essere sempre assegnato un valore prima che essa venga assegnata (definite assignment): il compilatore deve assicurarsi che ciò sia sempre verificato tramite data-flow analysis del codice.

I valori di default sono usati in generale per i tipi valore; ad esempio, in Java le variabili di classe, locali e i componenti di un array sono inizializzati al valore di default, ma non le variabili di istanza.

Il costruttore viene utilizzato per i tipi classe in Java, C++ e C#.

Per il clean up dello stato si usa un distruttore (o più propriamente finalizzatore) in C++ e C#; questo è unico, non ereditabile, senza overload, parametri e modificatori. È invocato automaticamente alla distruzione dell'oggetto. In Java invece si usa il metodo di Object **`finalize()`**; viene invocato automaticamente alla distruzione dell'oggetto e il momento in cui viene invocato dipende dalla JVM. Per liberare la memoria in C si usa la funzione free(), in C++ si usano le funzioni free() e delete mentre in Java e C# si usa il garbage collector.



Interfaccia vs Classe astratta

Interfaccia	Classe astratta
Deve descrivere una funzionalità semplice, implementabile da oggetti eterogenei (cioè appartenenti a classi non correlate tra di loro).	Può descrivere una funzionalità anche complessa, comune a un insieme di oggetti omogenei (cioè appartenenti a classi strettamente correlate tra di loro).
Può "ereditare" da 0 a più interfacce.	Può "ereditare" da 0 a più interfacce e da 0 a più classi (astratte e/o concrete): in questo caso deve essere almeno una classe se esiste una classe radice di sistema e al massimo una classe se non è ammessa l'ereditarietà multipla.
Non può essere istanziata.	Non può essere istanziata.
Non può contenere uno stato.	Può contenere uno stato (comune a tutte le sottoclassi).
Non può contenere attributi membro e metodi (e proprietà ed eventi) statici (a parte eventuali costanti comuni)	Può contenere attributi membro e metodi (e proprietà ed eventi) statici.
Non contiene alcuna implementazione.	Può essere implementata completamente, parzialmente o per niente.
Le classi concrete che la implementano devono realizzare tutte le funzionalità.	Le classi concrete che la estendono devono realizzare tutte le funzionalità non implementate e possono fornire una realizzazione alternativa a quelle implementate.
Deve essere stabile: se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate.	Può essere modificata. Quando si aggiunge un metodo a una classe astratta già in uso, è possibile fornire un'implementazione di default, in modo tale da non dover modificare le sottoclassi.
Non può gestire la creazione delle istanze delle classi che la implementano.	Può gestire la creazione delle istanze delle sue sottoclassi.
La creazione deve essere effettuata: • dai costruttori delle suddette classi; • da un'istanza di una classe non correlata, la cui unica funzionalità è la creazione di istanze di altre classi (classe factory).	La creazione può essere effettuata come per l'interfaccia, ma anche da un metodo statico della classe astratta (metodo factory).